Chapter 5

The Execution of Genetic Algorithms in the Prototyping Board

This chapter provides the pseudocode of genetic algorithms (GA) executing in the prototyping board. It was necessary to design a custom microprocessor performing GA efficiently with a minimum number of instructions. The GA presented here has a strong impact on the design of the instruction set. The first version of GA was written in C language which can be executed and debugged on a conventional workstation. Then the C version was manually rewritten in assembly language. An assembler for the microprocessor was implemented. The high-level language compiler was not implemented since it required a lot of time and the microprocessor was not intended for general purpose. Finally, the assembly version was compiled to machine code downloaded into an EEPROM on the prototyping board.

The main algorithm is shown in Algorithm 5.1. First, the initial population, consisting of P individuals, was created at random. The generation began with producing Q offspring using crossover. Next, the mutation was employed to produce R offspring. The P individuals were selected from $(P + Q + R)$ individuals to survive in the next generation. The while loop (Algorithm 5.1, line 3-8) was repeated until the optimal solution was found or the number of generations reached the maximum value set at 50,000. The P, Q, and R were set at 128, 256, 128 respectively. Note that the 2^n-numbers were intended for simple random number generation.

The input/output sequences, used in fitness evaluation, were generated using the Algorithm 5.2. A random input sequence was fed to the target circuit to get the corresponding output sequence. Each sequence always began at the start state of the target circuit. Then the input/output sequences were stored in an EEPROM placed on the prototyping board.

The first population was initialised using Algorithm 5.3. The P individuals were created at random. Each bit had an identical probability to be "0" or "1". An individual was assigned its fitness calculated using Algorithm 5.4. The individual executed one input, then gave one output (Algorithm 5.4, line 5) and changed to next state (Algorithm 5.4, line 6). The output, in line 5, was compared to the desired output, then the fitness was increased by the number of similar output bits (Algorithm 5.4, line 7).

The generation began with employing crossover to produce Q offspring (Algorithm 5.5). Two parents were selected randomly from P individuals surviving from the previous generation. The selected parents were performed single-point crossover using Algorithm 5.6, then the new offspring were assigned their fitness (Algorithm 5.5, line 6-7).

Next, R individuals were produced using mutation (Algorithm 5.7). The R individuals were randomly selected from P individuals surviving from the previous generation, then the selected individuals were mutated (Algorithm 5.8). Each bit of an individual was changed with the probability $P_m = 0.01$. Following that, the new individuals were assigned their fitness (Algorithm 5.7, line 3).

After reproduction, there were $(P + Q + R)$ individuals. Next, the P individuals were selected to the next generation while the $(Q + R)$ individuals were discarded. The selection was presented in Algorithm 5.9. First, the fittest individual was selected (Algorithm 5.9, line 1-4). The loop (Algorithm 5.9, line 6-40) was repeated $P - 1$ times to select the individuals which were fit and different from the others which had been selected. The individuals were sorted by fitness to create f_rank (Algorithm 5.9, line 8-15). The diversity of an individual was defined as the different bits between the individual and the selected individuals (Algorithm 5.9, line 17-20). Then the individuals were sorted by diversity to create d_rank (Algorithm 5.9, line 22-29). The c_rank was the sum of f_rank and d_rank (Algorithm 5.9, line 31-32). The best individual in c_rank was selected (Algorithm 5.9, line 37-39).

Algorithm 5.1 Main.

```
line  1: generation = 0;
line  2: initialise P individuals;
line  3: while termination conditions not met do
line  4:     produce Q individuals using crossover;
line  5:     produce R individuals using mutation;
line  6:     select  P individuals from (P+Q+R) individuals;
line  7:     generation = generation + 1;
line  8: endwhile
```

Algorithm 5.2 Generating input/output sequences.

```
Generating (m x n) input/output sequences, where m is the number of
sequences and n is sequence length.

target    denotes for the target circuit, which is an FSM.
inp[m][n] denotes for input sequences.
out[m][n] denotes for output sequences.

rand()    returns an integer random number between 0 and 2^31 - 1

f1(fsm, state, input) is a state-transition function of the fsm.
f2(fsm, state, input) is an output-mapping function of the fsm.

line  1: for (i = 1; i <= m; i++)
line  2:     current_state = start_state;
line  3:     for (j = 1; j <= n; j++)
line  4:         inp[i][j] = rand();
line  5:         out[i][j] = f2(target, current_state, inp[i][j]);
line  6:         current_state = f1(target, current_state, inp[i][j]);
line  7:     endfor
line  8: endfor
```

Algorithm 5.3 Initialising population.

```
individual[1..P][1..N] denotes for P individuals of which the lengths
                       equal N.
fitness[1..P]          denotes for the fitness of P individuals.
rand()                 returns an integer random number between
                       0 and 2^31 - 1.

line  1: for (i = 1; i <= P; i++)
line  2:     for (j = 1; j <= N; j++)
line  3:         individual[i][j] = rand() % 2;
line  4:     endfor
line  5:     fitness[i] = EvaluateFitness(individual[i]);
line  6: endfor
```

Algorithm 5.4 Fitness evaluation.

```
Given (m x n) input/output sequences, where m is the number of
sequences and n is sequence length.

idv          denotes for an individual which is being evaluate.
inp[M][N]    denotes for input sequences.
out[M][N]    denotes for output sequences.
fitness      denotes for the fitness of the individual.

f1(fsm, state, input) is a state-transition function of the fsm.
f2(fsm, state, input) is an output-mapping function of the fsm.
diff(str1, str2) returns the number of similar bits between
                 str1 and str2.

fitness = 0;
for (i = 1; i <= M; i++)
    current_state = start_state;
    for (j = 1; j <= N; j++)
        output = f2(idv, current_state, inp[i][j]);
        current_state = f1(idv, current_state, inp[i][j]);
        fitness = fitness + diff(out[i][j], output);
    endfor
endfor
```

Algorithm 5.5 Reproduction (crossover).

```
individual[1..P]       denotes for P individuals surviving from the
                       previous generation.
individual[P+1..P+Q]   denotes for Q individuals produced by crossover.
fitness[P+1..P+Q]      denotes for the fitness of Q individuals.
rand()                 returns an integer random number between
                       0 and 2^31 - 1.

for (i = P+1; i <= P+Q; i = i + 2)
    Crossover(individual[1 + (rand() % P)],
              individual[1 + (rand() % P)],
              individual[i],
              individual[i+1]);
    fitness[i] = EvaluateFitness(individual[i]);
    fitness[i + 1] = EvaluateFitness(individual[i + 1])
endfor
```

I19275596

Algorithm 5.6 Single-point crossover.

Given two parent selected randomly from P individuals surviving from the previous generation, the crossover produces two children.

`par1[1..n]`	denotes for parent1, which is a n-bit string.
`par2[1..n]`	denotes for parent2, which is a n-bit string.
`chd1[1..n]`	denotes for child1, which is a n-bit string.
`chd2[1..n]`	denotes for child2, which is a n-bit string.
`rand()`	returns an integer random number between 0 and 2^31 - 1.

```
cut_point = 1 + (rand() % n);
chd1[1..cut_point] = par1[1..cut_point];
chd1[(cut_point + 1)..n] = par2[(cut_point + 1)..n];
chd2[1..cut_point] = par2[1..cut_point];
chd2[(cut_point + 1)..n] = par1[(cut_point + 1)..n];
```

Algorithm 5.7 Reproduction (mutation).

`individual[1..P]`	denotes for P individuals surviving from the previous generation.
`individual[(P+Q+1)..(P+Q+R)]`	denotes for R individuals produced by mutation.
`fitness[(P+Q+1)..(P+Q+R)]`	denotes for the fitness of R individuals.
`rand()`	returns an integer random number between 0 and 2^31 - 1.

```
for (i = (P+Q+1); i <= (P+Q+R); i++)
    Mutation(individual[1 + (rand() % P)], individual[i]);
    fitness[i] = EvaluateFitness(individual[i]);
endfor
```

Algorithm 5.8 Mutation.

Given a parent selected randomly from P individuals surviving from the previous generation, the mutation produces a child.

`par[1..n]`	denotes for parent, which is a n-bit string.
`chd[1..n]`	denotes for child, which is a n-bit string.
`drand()`	returns floating-point values uniformly distributed between [0.0, 1.0).
`Pm`	denotes for mutation probability, set at 0.01.

```
for (i = 1; i <= n; i++)
    if (drand() < Pm)
        chd[i] = not par[i];
    else
        chd[i] = par[i];
endfor
```

Algorithm 5.9 Selection.

```
individual[1..(P+Q+R)] denotes for (P+Q+R) individuals.
fitness[1..(P+Q+R)]  denotes for fitness.
diversity[1..P+Q+R]  denotes for diversity.
f_rank[1..(P+Q+R)]   denotes for fitness rank.
d_rank[1..(P+Q+R)]   denotes for diversity rank.
c_rank[1..(P+Q+R)]   denotes for combined rank.

initialise diversity[1..(P+Q+R)] to zero.

find n, where n is an integer number between 1 and (P+Q+R)
        and fitness[n] is maximum;
swap(individual[n], individual[1]);
swap(fitness[n], fitness[1]);

for (i = 2; i <= P; i++)

   sort individual, fitness, diversity, f_rank, d_rank, c_rank
        from i to (P+Q+R) by fitness;

   n = 1;
   for (j = i; j <= (P+Q+R); j++)
       f_rank[j] = n;
       n = n + 1;
   endfor

   for (j = i; j <= (P+Q+R); j++)
       diversity[j] += number of different bits between
                       individual[i - 1] and individual[j];
   endfor

   sort individual, fitness, diversity, f_rank, d_rank, c_rank
        from i to (P+Q+R) by diversity;

   n = 1;
   for (j = i; j <= (P+Q+R); j++)
       d_rank[j] = n;
       n = n + 1;
   endfor

   for (j = i; j <= (P+Q+R); j++)
       c_rank[j] = f_rank[j] + d_rank[j];

   find n, where n is an integer number between i and (P+Q+R)
           and c_rank[n] is minimum;

   swap (individual[n], individual[i]);
   swap (fitness[n], fitness[i]);
   swap (diversity[n], diversity[i]);
endfor
```